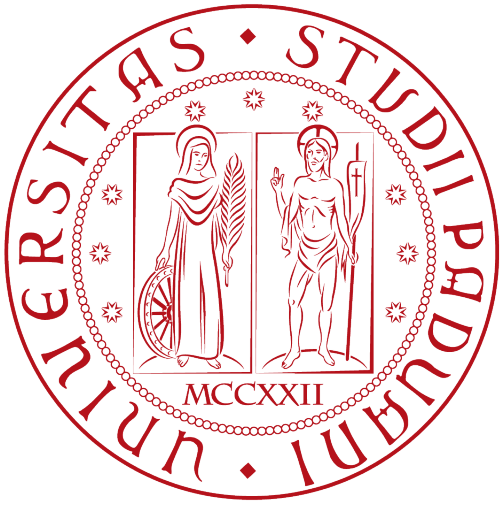

UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

PARITORRENT 2012 ANALISI FUNZIONALE E PRESTAZIONALE

RELATORE : Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi

LAUREANDO : Fincato Matteo

A.A. 2012 - 2013

Più studi, più sai...
Più sai, più dimentichi...
Più dimentichi, meno sai!
E allora chi te lo fa fare?
–Lupo Alberto–

Indice

Sommario

Uno dei protocolli peer-to-peer piu diffusi ai nostri giorni è BitTorrent, di cui esistono numerose implementazioni. Il plugin Torrent sviluppato all'interno del progetto PariPari e un client BitTorrent scritto in Java che supporta molte delle funzionalità fornite dai client piu comuni disponibili gratuitamente su Internet.
Questa tesi ha lo scopo di creare un manuale per i futuri pariparisti del plugin, affrontando la spiegazione del protocollo BitTorrent con alcune delle sue principali estensioni ed in particolare vedendone la sua implementazione nel plugin Torrent.
Si darà anche una breve introduzione del software e progetto PariPari.
Inoltre prevede un'analisi prestazionale del plugin rispetto ad altri client sulla rete, analisi putroppo non andata a buon termine. Se ne daranno quindi le spiegazioni del fallimento.

1. Introduzione a PariPari

PariPari è un progetto software attualmente in sviluppo presso il Dipartimento di Ingergneria dell'Informazione dell'Università di Padova.
Tale progetto ha come obiettivo la realizzazione di una rete peer-to-peer *serverless*, nella quale cioè non sono presenti nodi centrali (tutti i nodi che fanno parte della rete sono sia client che server), e di una piattaforma multifunzionale basata su servizi decentralizzati come ad esempio VoIP[1], peer-to-peer file sharing, instant messaging ed altri.
E' stato interamente scritto con il linguaggio di programmazione Java, permettendo così di essere disponibile per tutti i maggiori sistemi operativi grazie alla portabilità dell'ambiente stesso.
Il software è pensato con una struttura modulare, permettendo così una maggiore scalabilità alle esigenze del singolo utente e un più semplice sviluppo futuro di plugin. In questo modo un utente può così avere un client di PariPari totalmente personalizzato senza dover caricare tutti i plugin sviluppati ma solamente quelli che gli interessano.

Figura 1.1.: Il logo di PariPari.

[1]Voice over IP

2. La struttura di PariPari

La struttura di PariPari è modulare : ogni sua funzionalità è racchiusa in un modulo (o *plugin*).
Il modulo principale è il ***Core,*** che è il "cuore" del programma : esso gestisce le comunicazioni tra i vari plugin.
Escluso il plugin principale, tutti gli altri, per motivi funzionali e strutturali, sono suddivisi in due categorie : i plugin della cerchia interna e quelli della cerchia esterna.

2.1. La cerchia interna

Fanno parte della cerchia interna quei plugin che offrono servizi e risorse (tipo i socket TCP e UDP e spazio di memorizzazione) agli altri plugin.
I plugin di questa cerchia sono :

- **Connectivity :** Questo plugin offre e gestisce servizi di connettività, in particolare fornisce una vasta gamma di API per i socket, che sono necessari ai plugin della cerchia esterna per funzionare propriamente.

- **Credits :** Esso è integrato nel Core e gestisce il sistema di crediti presente nel software, che consiste nel "pagamento" di una "somma" tramite questi crediti (che possono idealmente essere pensati come delle monete) da parte dei plugin per richiedere delle risorse.

- **Local Storage :** Questo è un'altro plugin fondamentale per PariPari perchè gestisce la lettura e la scrittura di file, da parte degli altri plugin, sui dispostivi di memorizzazione di massa.

- **DHT** : Questo plugin implementa la *Distributed Hash Table,* che memorizza le informazioni degli altri utenti nella rete del client PariPari utilizzando una versione modificata di Kademlia.

2.2. La cerchia esterna

Questa cerchia comprende tutti i plugin che non occorrono al funzionamento base del software PariPari, che quindi possono essere aggiunti o rimossi, come si accennava nel capitolo 1, a seconda delle esigenze dell'utente.
Si elencano di seguito alcuni di questi.

- **Torrent** : E' il plugin che rappresenta il client per la rete BitTorrent, che verrà approfonditamente presentato più avanti.
- **Mulo** : Questo plugin rappresenta il client per la rete eMule e include molte funzionalità prensenti nei più comuni client disponibili sulla rete al giorno d'oggi ed altre di sua proprietà.
- **IM**[1] **e IRC**[2] : Questi due plugin sono entrambi client per il servizio di messaggistica istantanea (comunemente nota come *chat*). Sono entrambi basati sul protocollo *Jabber*, ora noto come XMPP[3], che offre una serie di servizi e protocolli gratuiti sviluppati ad hoc.
- **Web** : Il plugin rappresenta un server web distribuito. Ciò vuol dire che le pagine non sono salvate su una singola macchina ma sono distribuite su tutta la rete (PariPari, ovviamente) e le richieste sono soddisfatte da nodi appropiati.
- **GUI** : Questo racciude la *Graphical User Iterface* (l'interfaccia grafica), che permette un uso più semplice di PariPari agli utenti medi.

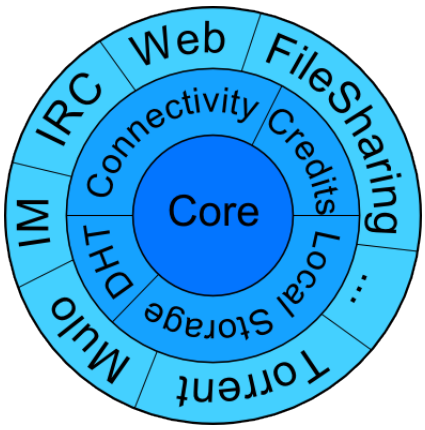

Figura 2.1.: Rappresentazione dei plugin con la suddivisione delle cerchie

[1]Istant Messaging
[2]Internet Relay Chat
[3]Extensible Messaging and Presence Protocol, `http://xmpp.org`

3. Il protocollo BitTorrent

In questo capitolo viene presentato il funzionamento di base del protocollo BitTorrent, per dare una panoramica sui concetti fondamentali di funzionamento sui quali si basa il plugin Torrent.

3.1. Descrizione generale

BitTorrent è un protocollo usato per la distribuzione di grandi moli di dati attraverso la rete internet. Nato dalla mente di Bram Cohen nel 2001, il quale ha progettato anche nello stesso anno il client noto con il nome BitTorrent, crebbe rapidamente e a tutt'oggi è un protocollo molto diffuso, tanto da istituire una compagnia, la *BitTorrent Inc*.

Esso si basa sul principio secondo il quale ogni utente (*client*) connesso alla rete non solo scarica i file, me ne diventa anche contemporanemante distributore. Infatti, mentre scarica, rende disponibili le parti dei file già scaricati e permette ad altri utenti di poterli scaricare a loro volta da lui stesso. In questo modo si vanno a suddividere le richieste per un file su più macchine alleggerendo cosi l'unica macchina che lo distribuirebbe (*server*), la quale, se si considerasse il caso peggiore di un file molto richiesto sulla rete, dovrebbe sopperire altrimenti a centinaia o addirittura migliaia di richieste arrivando a "rompersi," non distrbuendo così più il file!

Il procedimento che ogni utente deve seguire per scaricare un file (o una serie di file legati tra loro) tramite BitTorrent è il seguente:

1. Cercare su un BitTorrent index (siti web come torrentreactor.net) o altri siti web (tipo torrentz.eu che effettuano la ricerca sui precedenti), oppure ottenere direttamente dall'autore un file .torrent che si riferisce al contenuto desiderato

2. Scaricare questo file

3. Aprire il file con un client BitTorrent per procedere con il download

4. A download completato rimanere nella rete come seeder (comportamento opzionale ma di buona norma per la salute della rete)

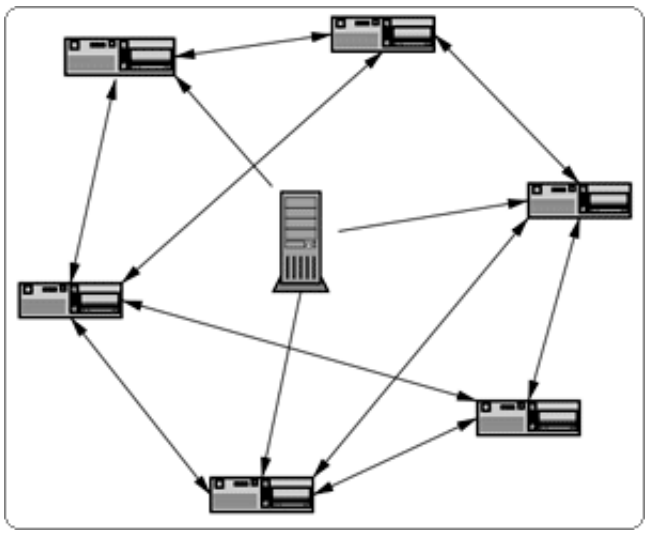

Figura 3.1.: La soluzione di BitTorrent

3.2. Il file metainfo

Un utente (*peer*) per poter scaricare uno o più file dalla rete deve possedere il file metainfo, che chiameremo *file torrent* dato che ha estensione .torrent, scaricandolo dal web. Tale file è necessario perchè non è possibile effettuare la ricerca dei file da scaricare all'interno della rete di BitTorrent, dato che questa non è indicizzata. All'interno del file torrent sono raccolte tutte le informazioni per effettuare il download, organizzate secondo coppie chiave-valore di un dizionario e codificate in BEncode.
Le chiavi principali sono :

- **announce :** l'URL del tracker; può anche essere una lista di URL di vari tracker da contattare.
- **info :** è un sotto-dizionario che contiene tutte le informazioni sui file da scaricare.

Tutte le stringhe nel file torrent che contengono testo devono essere codificate in UTF-8[1].

3.2.1. Il sotto-dizionario info

Nel sotto-dizionario info sono presenti le seguenti chiavi :

- **name :** tale chiave contiene la stringa che indentifica il nome del file da salvare (oppure il nome della directory nel caso siano più file).
- **piece length :** contiene il numero di byte di ciascun pezzo in cui il file viene diviso. Ai fini del trasferimento, i file sono suddivisi in pezzi di dimensione fissa tutti della

[1]La codifica UTF-8 è una codifica dei caratteri Unicode in sequenze di lunghezza variabile di byte (da 1 a 4) - `http://tools.ietf.org/html/rfc3629`.

stessa lunghezza a parte l'ultimo che può essere troncato. La lunghezza dei pezzi è solitamente una potenza di due, molto comunemente si usa 2^18 = 256 KB.

- **pieces** : tale chiave mappa una stringa di lunghezza multipla di 20. Viene suddivisa in sottotringhe di lunghezza 20 byte, ciascuna delle quali è l'hash SHA-1[2] del pezzo al corrispondente indice.

- **length** or **files** : deve esserci una sola di queste due chiavi. Se è presente *length* allora il download è composto di un solo file, altrimenti si ha un insieme di file da scaricare che si posizionano in una struttura di directory. Nel singolo caso *length* rappresenta la lunghezza del file in byte; nel caso del multi-file il valore di *files* costituisce una lista di file ed è composta da un dizionario contenente le seguenti chiavi :

  - **length** : rappresenta la lunghezza del file in byte
  - **path** : una lista di stringhe corrispondente ai nomi delle sottodirectory, l'ultima delle quali è il nome dell'attuale file (una lista vuota è un'errore).

3.3. Il tracker

Il tracker è un server utile a reperire le informazioni degli altri utenti (*peers*) che stanno scaricando lo stesso file torrent o perlomeno lo hanno in condivisione.
Esso risponde a richieste HTTP di tipo GET (preleva).
Quando un peer contatta il tracker, attraverso l'URL contenuto nell'announce nel file torrent, inviandogli un serie di informazioni (tra le quali l'infohash del torrent[3], il suo id,la porta sulla quale resta in ascolto, l'indirizzo ip, ecc.) questo risponde con una lista di al più 50 peer, scelti a caso tra le sue liste, che stanno appunto scaricando o caricando lo stesso file torrent.
A questo punto il peer può contattare i peer della lista ricevuta per iniziare il download.
Una volta terminato, il peer contatta il tracker avvisandolo del completamento; questo provvederà cosi ad inserirlo nella lista dei peer, associata a quel particolare file torrent, che hanno il file completo. Infatti il tracker mantiene inoltre delle statistiche di salute del file torrent, memorizzando quanti utenti che possiedono quel file in media sono presenti sulla rete. Questi utenti vengono suddivisi in due categorie : i *seeders,* ovvero sono i peer che hanno completato il file e che lo stanno condividendo, e i *leechers,* cioè quei peer che non hanno completato ancora il download. Un basso numero di seeders riduce la possibilità di riuscire a completare il donwnload.
E' usuale effettuare l'announce anche tramite il protocollo tracker UDP[4].

[2] Secure Hash Algorithm, è uno degli algoritmi di crittografia. - `http://tools.ietf.org/html/rfc3174`
[3] Composto da 20 byte che rappresentano l'hash SHA-1 del valore della chiave info nel file torrent, codificato in BEncode.
[4] Si veda `http://www.bittorrent.org/beps/bep_0015.html`

3.4. Peer wire protocol

Quando i peer devono comunicare tra loro devono rispettare un'insieme di regole e messaggi definite dal *Peer Wire Protocol.* Tale protocollo opera su TCP[5] o uTCP[6] e le connessioni tra i peer sono simmetriche: sia i messaggi che i dati possono viaggiare in entrambe le direzioni. Il protocollo consiste di un messaggio *handshake* iniziale, che i due peer si scambiano se entrambi hanno lo stesso file torrent, seguito da un flusso senza fine di messaggi di lunghezza prefissata. Successivamente si scambiano vari messaggi che vengono illustrati nel successivo sottoparagrafo.
L'handshake è costituito dal carattere diciannove (decimale) seguito dalla stringa 'BitTorrent protocol'. Il carattere iniziale indica la lunghezza della stringa.
Dopo l'header fisso si trovano 8 byte riservati, che servono per le estensioni del protocollo. Successivamente si trovano 20 byte che rappresentano l'hash SHA-1 del torrent (uguale al valore dell'*info_hash* presente nella richiesta GET inviata al tracker). Se entrambe le parti non inviano lo stesso valore, la connesione viene chiusa. Infine si trovano gli ultimi 20 byte che rappresentano l'id del peer, lo stesso inviato al tracker nel GET e contenuto nella lista di risposta inviata da quest'ultimo. Il peer che ha iniziato la connessione chiude quest'ultima se riceve un id diverso dall'id del peer che aveva contattato.
Questo è quanto avviene per l'handshake.
Il protocollo, inoltre, fa riferimeno ai pezzi dei file tramite indice (come già descritto nel file metainfo), partendo da zero.
Quando un peer ha finito il download di un pezzo ed ha controllato la corrispondenza dell'hash, annuncia il possesso di tale pezzo a tutti i suoi peer[7].
Ogni peer inoltre deve mantenere per ogni connessione due bit di stato[8] : ***choked*** (strozzato) o no, ***interested*** (interessato) o no. Il primo bit indica che non vengono inviati dati finchè non avviene un ***unchocked***. La connessione inizia con *chocked* e *not interested.* Per ulteriori approfondimenti a riguardo, si veda [1].

3.4.1. I messaggi

Si osservano qui gli altri messaggi che i peer utilizzano per le comunicazioni.
Essi sono nella forma <length prefix><message ID><payload>. Il prefisso che indica la lunghezza è codificato in 4 byte big-endian[9], mentre l'ID è un singolo byte decimale. Il payload dipende dal messaggio.

[5] Trasmission Control Protocol - Consiste in un protocollo di comunicazione a pacchetto tra computer di una rete, che si occupa del controllo di trasmissione. - `http://tools.ietf.org/html/rfc793#page-1`
[6] uTorrent Transport Protocol - `http://www.bittorrent.org/beps/bep_0029.html`
[7] Questo comportamento non è in realtà seguito da molti client, che preferiscono una politica più accorta nello spedire messaggi HAVE, è inutile ad esempio informare un peer che si ha un pezzo che lui stesso ha, dato che non ne sarà mai interessato. Questo tipo di approccio più selettivo garantisce fino al 50% di overhead in meno, sono possibili altre accortezze per ridurlo ulteriormente.
[8] Si dice *bit di stato* un bit che rappresenta una informazione, anche in base al suo valore (0 o 1).
[9] Big-endian e Little-endian sono due metodi usati per memorizzare i dati in memoria. La differenza tra i due consiste nell'ordine in cui sono memorizzati: il primo viene memorizzato dal bit più significativo al meno saignificativo; viceversa il secondo.

Esiste un messaggio di lunghezza zero e privo di ID, chiamato *keep-alive,* che viene inviato ogni due minuti per mantenere attiva la connessione tra due peer quando tra loro non c'è nessun'altro scambio di messaggi.
Essi possono essere :

- 0 - **choke:** viene inviato ad un peer per avvisarlo che si sta per interrompere l'upload con esso.
- 1 - **unchoke:** inviato ad un peer per segnalargli che si rende disponibile l'upload ad esso.
- 2 - **interested:** viene inviato da un peer quando è interessato ai pezzi che possiede un'altro.
- 3 - **not interested:** viceversa del precedente.
- 4 - **have:** contiene l'indice del pezzo che un peer ha appena finito di scaricare e che invia a tutti gli altri per avvisarli del suo possesso.
- 5 - **bitfield:** inviato come primo messaggio subito dopo l'handshake, al suo interno è presente un'array di bit nel quale ciascun indice corrisponde al rispettivo pezzo e viene settato a 1 se il peer possiede il pezzo altrimenti a 0; se il peer non possiede pezzi per il torrent il messaggio non viene inviato.
- 6 - **request:** rappresenta la richiesta di un pezzo; contiene un'indice, un'inizio e la lunghezza, dei quali gli ultimi due sono degli offset per il file
- 7 - **piece:** viene inviato in risposta al precedente ed è il dowload vero e proprio del pezzo; contiene l'indice, l'inizio e il pezzo.
- 8 - **cancel:** questo messaggio viene inviato da un peer che vuole interrompere il download del pezzo richiesto. Ha lo stesso payload di request.

I primi tre sono privi di payload.
In Tabella 3.1 vengono riportati i messaggi con la lunghezza e la loro struttura.

3.5. Le principali estensioni al protocollo BitTorrent

3.5.1. Extension Protocol e Peer Exchange

L'Extension protocol fa parte delle librerie di *libtorrent* ed è stato sviluppato con lo scopo di fornire uno strato di supporto agli sviluppatori futuri di estesioni al protocollo ufficiale, rendendo semplice l'aggiunta di estensioni senza interferire con il protocollo standard o con altri client che non supportano questa estensione o altre che si vogliano aggiungere.

Messaggio	Lunghezza (byte)	Id	Payload
keep-alive	0000	-	-
choke	0001	0	-
unchoke	0001	1	-
interested	0001	2	-
not interested	0001	3	-
have	0005	4	<piece index>
bitfield	0001+X	5	<bitfield>
request	0013	6	<index><begin><length>
piece	0009+X	7	<index>begin><block>
cancel	0013	8	<index><begin><length>

Tabella 3.1.: La struttura dei messaggi.

Per comunicare agli altri client il supporto a tale estensione, nell'hanshake si imposta un bit tra i byte riservati e precisamente il 20esimo bit da destra (si ricorda che il conteggio inizia da zero da destra).
Dopo che è stato attivato il supporto per il protocollo di estensione da parte di entrambi i peer, il client supporta un nuovo messaggio rispetto a quelli definiti dal protocollo ufficiale.

Name	Id
extended	20

Tabella 3.2.: Il nuovo messaggio supportato.

Questo messaggio viene inviato come ogni altro messaggio di BitTorrent, con un prefisso di 4 byte che identifica la lunghezza e un byte che identifica il messaggio (in questo caso il byte rappresenta il 20). All'inizio del payload si trova un byte che fa da identificatore. Questo identificatore si riferisce a diverse estensioni del messaggio : quando vale zero identifica l'handshake dell'estensione; mentre con un ID maggiore di zero si identifica un messaggio relativo a tutte le altre estensioni.
Per approfondimenti e la definizione del messaggio di handshake si veda [5].
Un'altra importante innovazione, presente nelle librerie di *libtorrent*, è data dal Peer Exchange (abbreviato con PEX) che permette ai peer di un torrent di scambiare liste di peer attivi direttamente tra di loro, allegerendo cosi il tracker. Infatti dopo una fase iniziale di bootstrap, cioè quando si comunica con il tracker, i peer dipendono solamente dagli altri peer che implementano lo stesso protocollo per scoprire nuovi peer.
Attualmente non ci sono standard ufficiali per il protocollo PEX, varie versioni sono state sviluppate e implementate in vari client. Le due principali sono: quella utilizzata in Vuze denominata *az_pex* e quella usata da libtorrent denominata *ut_pex*. In entrambe le implementazioni i peer vengono divisi in due gruppi: peer a cui ci si e connessi (*added*) e peer a cui ci si e disconnessi (*removed*). Nell'implementazione di libtorrent, ognuna di queste liste è a sua volta formata da due liste: alla prima appartengono i peer che utilizzano IPv4, alla seconda i peer che impiegano IPv6.

Si sono inoltre imposte alcune condizioni da rispettare per spedizione dei messaggi :

- In un messaggio non possono esserci piu di 50 peer di tipo added tra la lista IPv4 e IPv6; lo stesso vale per i peer di tipo removed, quindi ogni messaggio di peer exchange contiene al massimo 100 peer.
- Un messaggio di peer exchange non dev'essere inviato piu frequentemente di una volta al minuto.

Per la lettura di un'articolo si veda [6].

3.5.2. Fast Extension

La Fast Extension è una delle estensioni ufficiali del protocollo BitTorrent che raggruppa al suo interno varie estensioni, con lo scopo di rendere il protocollo più efficiente e, come suggerisce il nome, più veloce.
I messaggi che formano queste estesioni sono:

- *Have all*
- *Have none*
- *Suggest piece*
- *Reject request*
- *Allowed fast*

Queste estesioni vengono attivate impostando il terzo bit meno significativo dell'ultimo dei byte riservati nell'handshake.
L'estensione viene però abilitatà solo se entrambe le parti della comunicazione impostano ad 1 questo bit.
I messaggi che formano le Fast Extension hanno la stessa sintassi dei messaggi tradizionali del peer protocol ma hanno una semantica diversa. Tutti gli interi sono allo stesso modo codificati in 4 byte big-endian e i messaggi iniziano sempre con il prefisso che indica la lunghezza del messaggio. Dopo tale lunghezza (tranne per il messaggio di *keep alive*) si trova un byte, che rappresenta un codice identificativo del messaggio, e zero o più argomenti che dipendono dal byte identificativo.
Per ulteriori approfondimenti si veda [3] e [4].

3.5.3. Azureus Messaging Protocol

Si tratta di un protocollo a se stante alternativo al protocollo standard implementato e realizzato dagli sviluppatori del client Vuze[10].
E' caratterizzato da un insieme di messaggi aggiuntivi che funziona solo tra due client che implementano tale estensione. Per indicare che un client supporta tale protocollo, si setta il bit

[10] Azureus fino alla versione 2. E' un client BitTorrent scritto in linguaggio Java.

più significativo del primo byte tra quelli riservati nell'hanshake del protocollo di BitTorrent. Se entrambi i client di una connessione rivendicano il supporto al protocollo devono quindi inviare l'un l'altro un messaggio di AZ-handshake per indicare quali messaggi essi supportano.

Come per il protocollo standard, i messaggi iniziano con un intero codificato con 4 byte che identifica la lunghezza del messaggio. Si trovano poi ulteriori 4 byte per un intero che identifica la lunghezza della stringa ID; la stringa ID che identifica il messaggio; un byte per l'intero che rappresenta la versione del protocollo. Il payload è un dizionario codificato in BEncode.

Per approfondimenti [7] e [8].

3.5.4. Extension Negotiation Protocol

L'Extension Negotiation Protocol è stato sviluppato per permettere ai client di scegliere quale protocollo usare tra l'Extension Protocol (*LTEP*) e l'Azureus Messaging Protocol (*AZMP*) dato che essi sono mutuamente esclusivi.

Il protocollo usa due bit riservati dell'handshake di BitTorrent e precisamente i bit 47 e 48. Se il client indica che non supporta nessuna delle due estensioni, allora il valore di qeusti due bit viene ignorato. Se invece un client indica il supporto ad AZMP, ma non ad LTEP, il valore dei bit viene ignorato anche questa volta e viene usato AZMP (se entrambi indicano il supporto ad esso). Al contrario, se un client indica il supporto a LTEP, ma non ad AZMP, il valore dei due bit viene ancora una volta ignorato e viene usato LTEP (se entrambi indicano il supporto ad esso).

Se inevece entrambi i client indicano il supporto ad entrambi i protocolli, allora il valore dei bit indica quale estensione del protocollo viene usata.

Viene qui riportato un elenco del valore in binario dei bit con la relativa assunzione fatta dai client per la comunicazione.

- **00** ***Force LTEP*** - Significa che il client non è a conoscenza del protocollo di negoziazione oppure che entrambi stanno forzando l'altro ad usare LTEP .

- **01** ***Prefer LTEP*** - Significa che il client preferisce LTEP ed userà AZMP solo se l'altro client indica che vuole forzare l'uso di tale protocollo.

- **10** ***Prefer AZMP*** - Indica che il client preferisce AZMP ed userà AZMP se l'altro client indica che preferisce AZMP o forza l'uso di AZMP.

- **11** ***Force AZMP*** - Significa che forza l'uso di AZMP e lo userà finchè non forzerà l'uso di LTEP (o non sono cosapeli dell'uso del negotiation protocol).

Si può dire che *Force "sovrascrive" Prefer* e che *LTEP "sovrascrive" AZMP.*

3.5.5. Protocol encryption

Dato che BitTorrent occupa una grande parte del traffico internet e viene sempre più usato per il download di materiale protetto da copyright, gli ISP[11] hanno iniziato a limitare il traffico prodotto da esso.
A tale scopo è stato realizzato il protocollo di encryption per offrire ai client un mascheramento dei pacchetti inviati, codificando il loro header ed opzionalmente anche il payload, in maniera tale che sembri un'invio di byte sul TCP completamente casuale. Quando è usato in unione a metodi di cifratura più forti (es. RC4[12]) esso fornisce anche un livello di sicurezza ragionevole per il contenuto incapsulato contro lo sniffing passivo dei dati.
Per ottenere un aspetto casuale sin dal primo byte il protocollo adotta uno scambio di chiavi di tipo Die-Hellman[13] che già di per sé usa interi casuali molto grandi; la fase successiva, ovvero la negoziazione della cifratura del payload, è essa stessa crittata e quindi in prima approssimazione sembra anch'essa casuale. Sono stati inseriti vari padding ad ogni fase in modo da scansare semplici analisi basate sulla lunghezza dei pattern.
Tale protocollo è indipendente dall'incapsulamento dei dati e quindi potrebbe essere usato per protocolli di livello superiore al TCP, ma è stato pensato per essere usato con il protocollo BitTorrent, con il protocollo AZMP o, in futuro, con Enhanced Messaging Protocol.
Per ora sono previsti due algoritmi di codifica del payload che sono il *plaintext* ed *RC4*, che forniscono diversi gradi di offuscamento, sicurezza e velocità di elaborazione: il plaintext consiste nel non usare alcun metodo per la cifratura, per cui il livello di oscuramento del traffico è solo quello dell'header, ovvero è un livello base e l'utilizzo della CPU è ridotto; al contrario RC4 offusca l'intero flusso dati e dà qualche sicurezza in più ma richiede un maggiore tempo di elaborazione e quindi maggior carico di CPU. E' bene sottolineare che lo scopo del protocollo è di realizzare l'offuscamento attraverso l'applicazione di tecniche crittografiche semplici, veloci ed efficienti: non si intende quindi riproporre un meccanismo robusto come SSL[14] per assicurare protezione completa al livello di trasporto, ed ecco perché ad esempio, non è stato adottato (per ora, per lo meno) un algoritmo di cifratura come AES[15].
Maggiori informazioni a riguardo di tale protocollo possono essere trovate in [13].

[11] Internet Service Provider, le compagnie che offrono, dietro la stipulazione di un contratto, servizi inerenti internet.
[12] E' un'algoritmo di cifratura dei dati. - `http://en.wikipedia.org/wiki/RC4`.
[13] Metodo specifico per lo scambio di chiavi di crittografia `http://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange`.
[14] Secure Socket Layer, protocollo crittografico che permette una comunicazione sicura ed assicura l'integrità dei dati su reti TCP/IP, agendo a livello di trasporto.
[15] Advanced Encryption Standard, algoritmo di cifratura a blocchi utilizzato come standard dal governo degli Stati Uniti d'America.

4. Implementazione di BitTorrent : Il plugin Torrent

In questo capitolo viene esposta la struttura e il funzionamento del plugin, come è stato implementato il protocollo BitTorrent, con alcune delle estensioni, per permettere al plugin di rappresentare un client e partecipare cosi alla relativa rete.
Il plugin Torrent è un modulo facente parte dei plugin della cerchia esterna che si occupa della gestione dei file torrent. E' un client BitTorrent conforme a tutte le specifiche del protocollo descritte nel Capitolo 3 a pagina 5 ed implementa anche tutte le estensioni viste (e altre!).

4.1. Il file di configurazione

Il file di configurazione è uno strumeto utile per settare dei parametri senza doverli impostare tramite modifica del codice.
Per apportare modifiche ai vari parametri, utili a Torrent per la sua esecuzione, è sufficiente esaminare il file `TorrentConfig.xml` (che si trova, all'interno della cartella *conf* nella cartella di *torrent* in *PariPariExperimental*) ed una volta fatto salvare il file e riavviare PariPari. Le modifiche così effettuate vengono immediatamente "raccolte" senza aver dovuto mettere mano al codice e doverlo ricompilare. Questo risulta utile per i futuri utilizzatori del programma che potranno così modificare le impostazioni di Torrent, quali per esempio il numero massimo di download simultanei, durante la sua esecuzione tramite la GUI salvandoli così nel file xml che verrà poi riletto alla prossima riesecuzione.
Il file viene comunque generato da codice perchè se non presente viene creato alla prima esecuzione ed i parametri impostati con dei valori di default.
La classe che si occupa della gestione di tale file è la `XMLConfigHandler` che si trova nel package `torrent.config`.
Al suo interno sono definite varie costanti che rappresentano i vari parametri ed ha il compito di gestire il file, leggendo e scrivendo i parametri invocando rispettivamente i metodi `readConfigFromFile()` e `writeConfigToFile()`.
Inoltre all'avvio di Torrent, quando viene invocato il metodo `initialize()`, crea la directory che conterrà il file, se questa non esiste già, chiamando il metodo `mkdir()` della classe FileAPI (si vedrà alla sezione 4.2 come avviene la comunicazione con i plugin della cerchia interna, in questo caso con *LocalStorage*) ed un file di configurazione con i parametri impostati a valori di default, a meno che questo non esista già ed allora si leggono da esso i valori dei parametri invocando il metodo `readConfigFromFile()` locale alla classe.
Se si vuole aggiungere un parametro che deve essere letto ad ogni avvio del plugin e che serve per una sua configurazione, sarà necessario aggiungere la relativa costante tra le altre presenti

ed all'interno del metodo `createDefaultConfig()` aggiungere il settaggio del parametro ad un valore di default inserendo semplicemente la chiamata al metodo `setProperty()` sull'oggetto `default` (oggetto di tipo *Properties* della classe *java.util*) per far si che questo venga scritto sul file XML la prima volta e poi possa così essere letto e risettato a piacere.

```
1 private final String TRK_PORT = "tracker_port";
2
3 ...
4
5 defaults.setProperty(TRK_PORT , Integer.toString(7881));
```

Codice 4.1: Definizione e settaggio di default di un parametro di esempio.

4.2. La comunicazione con i plugin della cerchia interna

Nel Capitolo 2 è stata evidenziata la struttura modulare di *PariPari* affermando che il core si occupa di gestire le comunicazioni tra i vari plugin.
In altre parole il *Core* si occupa di prendere in gestione le varie richieste effettuate dai plugin della cerchia esterna e smistarle ai plugin della cerchia interna avendo cosi sempre sotto controllo le risorse istanziate.
Torrent in particolare ha bisogno di accedere ai file su disco gestiti da *LocalStorage*,di socket gestiti da *Connectivity* e di crediti gestiti da *Credits* (plugin ancora in fase di sperimentazione). Questo può realizzarlo effettuando una chiamata al metodo del Core `askTheCore()`, il quale richiede come parametri la classe della quale si chiede la risorsa, i parametri da passare al suo costruttore ed un parametro booleano che indica se la risorsa deve essere automaticamente rinnovata dal *Core*.

Ad esempio se si volesse richiedere l'accesso ad un file si dovrebbe eseguire la chiamata illustrata nel Listato 4.5, dove *sender* è un oggetto *IPlugin*, interfaccia definita all'interno del *Core* che definisce vari metodi tra i quali anche il metodo sopraccitato.

```
1 RandomAccessFileAPI qraf = sender.askTheCore(
2         RandomAccessFileAPI.class,
3         new ConstructorParameters(
4                 featureValues,
5                 isf,
6                 AccessMode.RW),
7         true);
```

Codice 4.2: Utilizzo del metodo askTheCore.

Questo è quello che semplicemente deve fare *Torrent* quando deve interagire con i pugin che stanno al di sotto di esso.
Per ulteriori chiarimenti in merito si rimanda il lettore alla documentazione online di *PariPari* [14].

4.3. Il package principale

Il package principale è `torrent` e contiene 5 classi:

- TorrentCore
- TorrentConsole
- TorrentLogger
- TorrentListeners
- TorrentID

La più importante delle 5 è `TorrentCore` che, come si può capire dal nome, ha il compito di gestire il plugin.
Infatti al suo interno si trova il metodo `init()`, che viene eseguito all'avvio di *Torrent* e che crea tutte le istanze degli oggetti necessari per la sua l'esecuzione, e tutti i metodi per controllare lo scaricamento del file torrent (`addDownload()`, `startDownload()`, `stopDownload()`, `reloadDownload()`, `removeDownload()`, `getDownload()`).
La classe `TorrentConsole` invece interpreta ed esegue l'azione indicata dal relativo comando, tramite i metodi `parseMessage()` e `performAction()`, dato in input dall'utente attraverso la console di *PariPari*. Lo sviluppatore che volesse aggiungere un nuovo comando deve inserire il relativo codice da eseguire, quando il comando verra invocato tramite console, nel metodo `performAction()` ed inoltre all'interno del metodo `initHelp()` mettere l'aggiunta del comando tramite la chiamata al metodo `addCommand()` della stessa classe, il quale provvederà in fase di esecuzione ad inserire il comando nella lista dei comandi disponibili.

```
1 addCommand("ls", "show download queue");
```

Codice 4.3: Esempio di aggiunta alla console di un comando.

Per quanto riguarda le ultime tre classi, la prima si occupa di loggare eventuali messaggi sulla console di *PariPari*, la seconda gestisce i messaggi da e per il core di PariPari e l'ultima invece serve per creare l'ID che viene assegnato a ciascun torrent per la gestione di quest'ultimi all'interno del programma.

4.4. La classe Peer

I peer sono rappresentati tramite la classe Peer che si trova all'interno del package `torrent.peer`.
Tale classe serve per memorizzare tutte le informazioni relative ai peer, quali per esempio se è *interested*, *chocked*, il suo id, ip e porta, le informazioni relative alle estensioni, e molto altro.

4.5. Il download di un torrent

Il funzionamento di base è il seguente: una volta avviato *PariPari*, l'utente aggiunge il plugin *Torrent* attraverso la console con il comando `add torrent` oppure cliccando sull'apposita tab che trova in alto, che provoca il caricamento del file Jar di *Torrent* ed avvia `TorrentCore`. A questo punto, l'utente può procedere a ricaricare dei download pre-esistenti oppure aggiungerne dei nuovi cliccando sull'apposito bottone, tramite il relativo file torrent salvato su disco oppure direttamente tramite URL. Una volta che il file torrent è stato aggiunto (eseguendo il metodo `addDownload()` in `TorrentCore` che crea la relativa istanza di `DownloadTorrent`), l'utente può far partire il download digitando sulla console `start <index>` (dove index è l'indice del torrent visualizzato dal comando `ls`) che provoca la chiamata al metodo `startDownload()` che avvia il thread di `DownloadTorrent`, il quale crea l'istanza di `TorrentMessageSender` e `TorrentMessageReceiver` che, insieme all'istanza di `HTTPConnection`, inizia a contattare i tracker e successivamente i peer per scaricare/caricare il torrent.

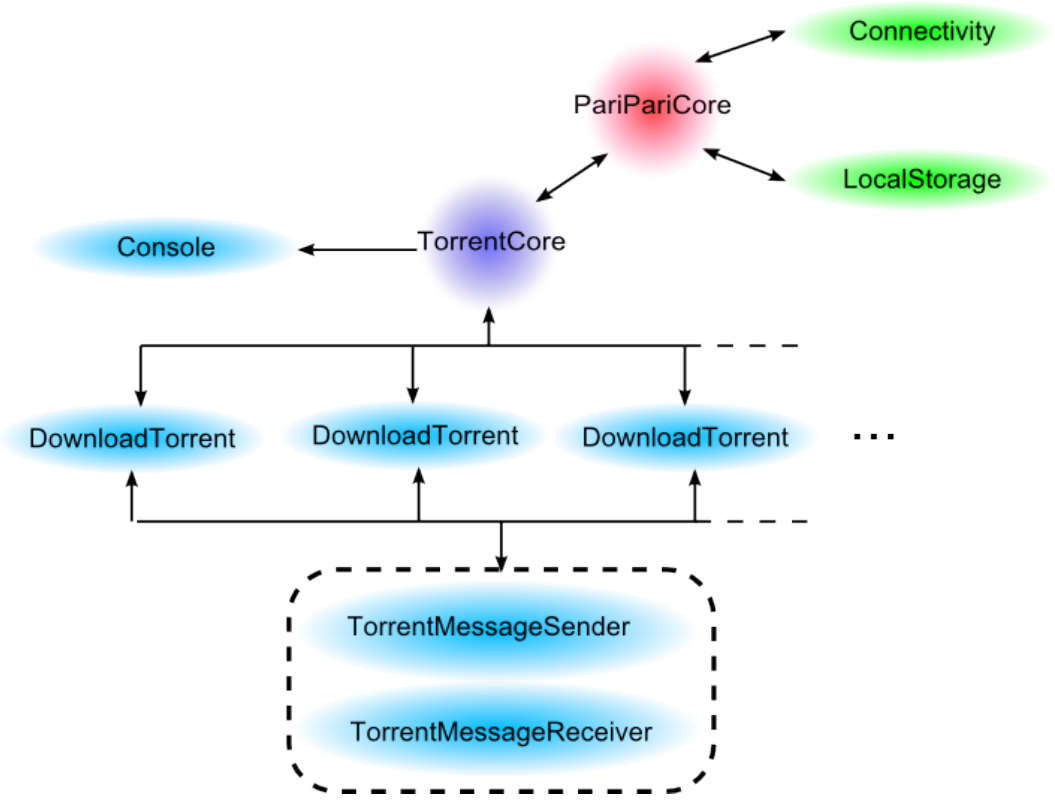

Figura 4.1.: Schema di funzionamento base del plugin.

4.5.1. Il thread principale: DownloadTorrent

Come si è spiegato, quando viene caricato il file torrent viene creata l'istanza della classe `DownloadTorrent` che costituisce il thread principale che gestisce completamente il do-

wnload. Esso si occupa infatti di controllare le connessioni con i peer e la messaggistica scambiata con essi, ma non solo: controlla il thread che contatta il tracker per aggiornare la lista dei peer, realizza l'unchoking e l'optimistic unchoking, gestisce i pezzi, e altro ancora.
La classe si trova all'interno del package `torrent.manager` ed al suo interno, data la sua mansione, sono presenti numerosi metodi. Se ne riportano alcuni che si ritiene essere tra i principali, dandone per ciscuno una breve descrizione :

- `startDownload()` : L'effetivo metodo che fà partire il download del file torrent.
- `startTrackerUpdate()` : Crea e fà partire un thread di `PeerListUpdater`[1] per ricevere e tenere aggiornata la lista dei peer dal tracker.
- `stopTrackerUpdate()` : Ferma il thread creato con la chiamata al metodo precedente.
- `updatePeerList(Map<String,IPeer> list)` : Aggiorna la lista locale dei peer aggiungendogli eventuali nuovi peer passati come parametro in una lista ed eventualmente si collega ad essi (invocando il metodo `initConnection()`).
- `initConnection(IPeer p, boolean initiate)` : Crea il socket e lo utilizza per connettersi al peer passato come parametro.
- `disconnect(IPeer p)` : Rimuove il task associato al peer e chiude il socket.
- `pieceCompleted(String peerID, int pieceNB, boolean complete)` : Serve per la gestione del salvataggio del pezzo e la comunicazione agli altri peer della sua possessione con il messaggio HAVE. Se è corrotto invece lo elimina e lo resetta per il successivo riscaricamento.
- `parseMessageReceived(IPeer p, Message m)` : Decodifica il messaggio (passato come parametro) ricevuto dal peer ed esegue la relativa azione associata.

Per ulteriori dettagli in merito al codice dei metodi e all'analisi di altri, si rimanda alla visione dell'intero codice della classe.

4.5.2. La connessione con il tracker

Quando un torrent viene aggiunto ed avviato, la prima operazione che viene effettuata è il collegamento con il tracker per richiedere la lista dei peer, come da specifica del protocollo BitTorrent. Successivamente, durante il download, viene mantenuta la connessione con il tracker per aggiornare periodicamente la suddetta lista.
Il package contenente le classi che realizzano tutto ciò è `torrent.tracker`. Al suo interno si trovano due classi:

[1] Vedi Sezione 4.5.2.

- `HTTPConnection` : Rappresenta la connessione con il tracker; al suo interno si trovano i metodi per la connessione con esso (`openConnection()` e `closeConnection()`), per inviare una richiesta GET (`sendRequest()`) e per leggere lo stream ricevuto (`getInputStream()`).

- `PeerListUpdater` : Delinea il thread che mantiene aggiornata la lista dei peer contattando, tramite un oggetto della classe precedente, il tracker (`contactTracker()`) e processando la mappa che riceve come riposta, che può essere sia un messaggio di errore che una lista di peer con delle informazioni come ad esempio l'intervallo che bisogna aspettare prima di ricontattarlo (`processResponse()`). Al suo interno si trova il riferimento al relativo listener (riferimento al `DownloadTorrent`, dato che implementa l'interfaccia `IPeerListUpdaterListener`, che gli è stato passato quando viene eseguito il metodo `startTrackerUpdate()`) che viene richiamato all'interno del metodo `fireUpdatePeerList()`, dopo il termine dell'esecuzione di `processResponse()`, per aggiornare la lista dei suoi peer passandogli la lista ricevuta dal tracker come parametro del metodo `updatePeerList()`.

4.5.3. Lo scambio di messaggi tra peer

Viene qui spiegato come avviene l'invio e la ricezione dei messaggi del protocollo standard di BitTorrent.
Innazitutto si deve dare una precisazione sulla classe `Peer`, la quale contiene tre oggetti fondamentali per il processo di scambio: un buffer, che viene usato per la ricezione, e due code, una per contenere i messaggi in ingresso e una per contenere quelli in uscita.
Lo scambio dei messaggi avviene grazie ai due thread di `TorrentMessageSender` e `TorrentMessageReceiver`, istanziati da `TorrentCore` al suo avvio ed avviati ad un primo peer che viene registrato su di essi (tramite la chiamata a `registerPeer()`) e cioè all'interno del metodo `initConnection()`. Come è stato rappresentato nella Figura 4.1 i due thread sono comuni a tutti i `DownloadTorrent` cosicché si ha un risparmio di memoria.
Il `TorrentMessageReceiver`, come si può capire dal nome, si occupa della ricezione dei messaggi ed infatti durante il suo cilco di vita controlla ciclicamente tutti i socket dei peer e se sul buffer di un socket ci sono dati, li passa sul buffer del peer (`read()`), ne analizza il conenuto (`analizeBuffer()`) creando il messaggio corretto(tramite i metodi `readPP()` e `readHS()`) e lo aggiunge nella relativa coda dei messaggi ricevuti all'interno del peer corretto per andare successivamente a decodificarli tramite il metodo `parseMessage()` presente in `DownloadTorrent` (azione possibile tramite il peer che contiene il riferimento ad esso).
Il thread di `TorrentMessageSender` invece, serve per inviare i messaggi. Infatti durante la sua esecuzione controlla ciclicamente se ci sono messaggi da inviare presenti nelle code di ciascun peer. Quindi per ciascun peer svuota la sua coda dei messaggi in uscita, pescando di volta in volta il prossimo messaggio in testa alla coda ed inviandolo invocando il metodo `sendMessage()`, il quale va a scrivere sul socket del peer associato i byte del messaggio che poi saranno inviati.
Le due classi si trovano nel package `torrent.messages`.

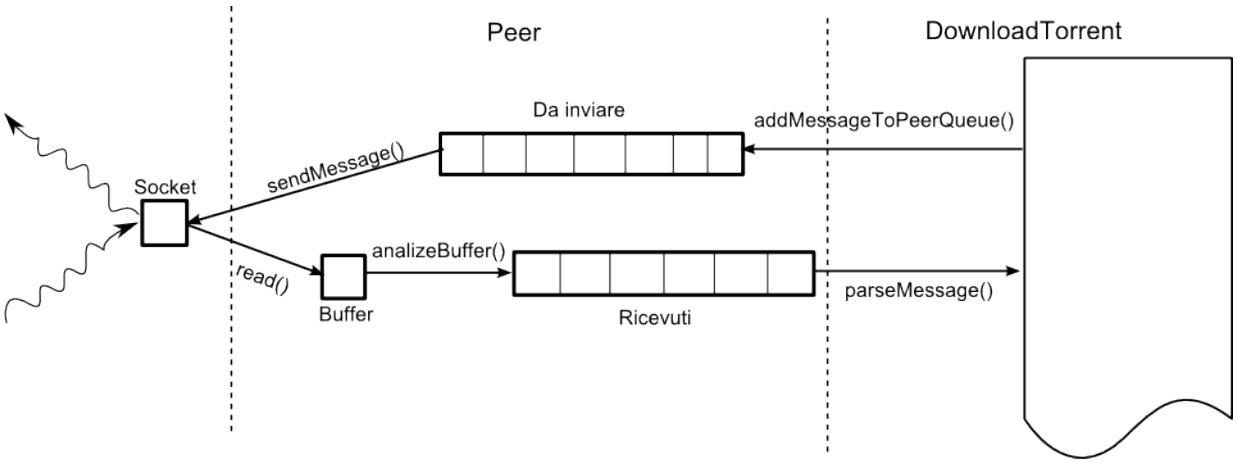

Figura 4.2.: Invio e ricezione dei messaggi.

All'interno del package `torrent.peer.messages` si trovano invece le classi che definiscono i messaggi del *Peer Protocol* : `MessagePP`, `MessageHS`.

La prima rappresenta tutti i messaggi del protocollo, ad esclusione dell'handshake che viene identificato nella seconda classe. Questa distinzione è stata fatta perchè l'hadshake ha un formato diverso dai restanti messaggi, proprio come è specificato nella documentazione di *BitTorrent*. Infatti all'interno delle classi si trovano variabili diverse che rappresentano i vari campi dei relativi messaggi. Entrambe le classi estendono la classe astratta `Message`, definita all'interno del package `torrent.peer.messages.interfaces`, che definisce la struttura generale che devono avere tutti i messaggi del protocollo standard e al loro interno si trova in particolare, tra gli altri, il metodo `generate()` che crea il messaggio sotto forma di stream di byte.

Nello stesso package si trova la classe `MessageFactory` che si occupa di creare i messaggi corretti nei vari casi. Quando si invia un messaggio si invoca il metodo `buildMessage()`, presente al suo interno, che creerà i messaggi istanziando oggetti delle classi precedenti.

```
1 this.tms.addMessageToPeerQueue(
2         p,
3         ((Peer) p).msgFactory.buildMessage(
4                 PeerProtocol.BITFIELD,
5                 this.fm.getCompleteBitField())
6         );
```

Codice 4.4: Esempio di aggiunta di un messaggio alla coda.

4.6. Le features implementate

4.6.1. Supporto al multitracker

Questa è una estensione minore ma è utile e molto usata. Infatti è molto comune al giorno d'oggi inserire più di un URL del tracker all'interno del file torrent, così se il primo tracker contattato non dovesse funzionare, il client può passare a provare a contattare il successivo nella lista, e così via.
A livello di codice tale features può essere vista nel metodo `contactTracker()` della classe `PeerListUpdater`, il quale va a pescare l'URL da un'array di stringhe, popolato con gli URL contenuti nella lista all'interno del file torrent.

4.6.2. Extension protocol e Peer Exchange

Il modulo *Torrent* supporta entrambe queste estensioni, presentate nella Sezione 3.5.1, perché è molto importante riuscire a contattare un numero elevato di peer per poter così raggiungere un'elevata velocità di download.
Le classi che definiscono i messaggi dei due protocolli si trovano nel package `torrent.peer.messages.ltep` e tutte estendono la classe `Message_LT`, che definisce la struttura standard dei messaggi del protocollo, la quale a sua volta estende la classe `Message` (per rendere i messaggi compatibili con gli altri messaggi del protocollo standard).
All'interno della classe `Message_LT` si trova, tra gli altri metodi, il metodo `generate()`, che si occupa di inserire come prefisso del contenuto dei messaggi le opportune intestazioni.
Per ogni tipo di messaggio e stata creata una classe apposita, contenente, tra i vari metodi, `encodePayload()`, `decodePayload()` e `buildPayload()`, che si occupano di codicare (per l'invio) , decodicare (per la ricezione) e creare la codifica BEncode del contenuto vero e proprio del messaggio. L'invio e la ricezione dei messaggi viene sempre gestito da `TorrentMessageSender` e `TorrentMessageReceiver`.
Le classi che implementano il *Peer Exchange* sono situate nel package `torrent.ltep` e si chiamano `UtPeerExchangeManager` e `UtPeerExchangeSender` che si occupano, ripettivamente, di gestire i peer ai quali sono connessi e la generazione e l'invio di messaggi di tipo *Peer Exchange*.
`UtPeerExchangeManager` contiene le strutture dati necessarie per il peer exchange ovvero una mappa di peer, contenente i peer ricevuti dagli altri, e quattro liste concatenate: due per le connessioni aggiunte (una per i peer che utilizzano IPv4[2] e una per i peer che usano IPv6[3]) e due per le connessioni chiuse (una per i peer che utilizzano IPv4 e una per i peer che utilizzano IPv6). I metodi `notifyDropped()` e `notifyAdded()` si occupano di inserire nelle liste opportune i peer, mentre i metodi `getNewlyAddedPeers()` e `getNewlyDroppedPeers()` gestiscono la mappa di peer, identificando i nodi di volta in volta aggiunti o rimossi.

[2] Quarta revisione dell'Internet Protocol.
[3] E' la versione dell'Internet Protocol designata come successiva alla quarta, presentata nel 1998 per far fronte alla saturazione di IPv4. Infatti essa permetterebbe un più vasto indirizzamento. Tuttavia nel 2011 risulta meno utilizzata della versione 4.

`UtPeerExchangeSender` è un thread che si occupa di creare, ogni 60 secondi, un messaggio di tipo peer exchange da inviare ai nodi a cui si è connessi in quel momento.
Si fa notare che i peer contenuti nei messaggi di tipo ut_pex sono rappresentati in forma compatta (cioe come array di byte), non come liste di tipo BEValue (cosa che avviene invece per i messaggi peer exchange di Azureus).

4.6.3. Fast Extension

Nella Sezione 3.5.2 è stata data una veloce descrizione di tale protocollo.
I messaggi sono stati implementati ciascuno in una singola classe all'interno del package `torrent.peer.messages.fastextension`. Come per tutti gli altri messaggi visti, le classi estendono la classe `Message` ed hanno al loro interno, tra gli altri, il metodo `generate()`.

4.6.4. Azureus Messaging Protocol

Il nostro modulo può comunicare anche con i client *Vuze* dato che gli è stato implementato il relativo protocollo.
Per i messaggi sono state realizzate delle classi apposite che sono allocate nel package `torrent.peer.messages.azmp`. Come per il protocollo standard, è sata costruita la classe `Message_AZ` che rappresenta il messaggio base del protocollo, la quale estende la classe `Message` ed implementa così il metodo `genarate()`. Tutte le altre classi che costituiscono gli altri messaggi specifici del protocollo estendono questa classe e realizzano, tra gli altri, i metodi `encodePayload()` e `decodePayload()` che, rispettivamente, codificano e decodificano il payload e setta, per quanto riguarda il decode, correttamente i vari campi del messaggio associato.
Per l'invio e la ricezione dei messaggi si utilizzano le classi viste nel protocollo standard avendo apportato le oppurtune modifiche. In particolare nella `MessageFactory` è presente il metodo `enableAzmp()` che serve per abilitare il protocollo AZMP e il metodo `buildMessage()` può costuire, se è stato abilitato il protocollo tramite il precedente metodo, anche i messaggi di tale protocollo. Invece in `TorrentMessageReceiver` si trova il metodo `readAZ()` che legge appunto un messaggio di AZMP che verrà invocato da `analizeBuffer()`. Per inviare messaggi non ci sono metodi dedicati, viene usato il solito metodo `addMessageToQueue()` che poi invocherà il metodo `sendMessage()` (che provocherà l'effettivo invio), passandogli un messaggio correttamente costruito tramite `buildMessage()`. Infine in `DowloadTorrent` sono presenti i metodi `parseAzMessage()`, `sendAzHandshake()` e `sendAzBadPiece()`, che dalla loro firma si capisce cosa fanno, e in `parseMessageReceived()` si trovano le sezioni dedicate alla decodifica dei messaggi di AZMP.
Il protocollo di Azureus include inoltre il peer exchange. La sua implementazione nel plugin si trova nel package `torrent.peer.messages.azmp.peerExchange`. All'interno di questo package si trovano due classi: `PeerExchangeManager` e `PeerExchangeSender`, con analogo comportamento delle rispettive classi di LTEP.

4.6.5. Extension Negotiation Protocol

Questa semplice estensione permette al client di poter scegliere quale delle due estensioni usare tra *AZMP* e *LTEP*. Come gia descritto nella Sezione 3.5.4, funziona usando i due bit riservati dell'hashake. In base alla combinazione di quesi due bit, il client abilita un protocollo o l'altro dopo essersi accordato con l'altro client della connessione.
Tutto questo avviene quando riceviamo il messaggio di handshake, all'interno della classe `DownloadTorrent` precisamente nel metodo `parseMessageReceived()`, ed invochiamo il metodo `setPeerExtensionsSupport()` che, passato l'handshake, abilita il supporto all'estensione e la sceglie.

4.6.6. Protocol Encryption

L'implementazione della crottografia all'interno del modulo Torrent è stata realizzata in due classi che si trovano nel package `torrent.crypto`: `RC4Engine` e `EncryptionManager`.
La prima classe è una realizzazione dell'algoritmo RC4. Infatti al suo interno troviamo, tra gli altri, metodi per crittare o decrittare (`encrypt()`, `decrypt()`) singoli byte o array di byte da inviare o ricevere e il metodo principale `processByte()` che si occupa di realizzare l'algoritmo vero e proprio.
La seconda classe si occupa di gestire in generale i parametri per la crittografia dettati dalle specifiche. Vengono definiti al suo interno vari metodi, tra i quali: metodi per la generazione della chiave pubblica D-H (`generateMyPubKey()`, che sfrutta le classi messe a disposizione dalla JCA[4]), per la memorizzazione della chiave ricevuta dal peer remoto e per l'inizializzazione ed utilizzazione dei motori di cifratura RC4 (sfruttando i metodi definiti nell'altra classe).
Per la gestione dell'handshake crittato, è presente nel package `torrent.peer.messages` la classe `Message_ENC_HS` che viene usato da `TorrentMessageSender` e `TorrentMessageReceiver` durante la fase iniziale di negoziazione.

[4] Java Cryptography Architecture, libreria inclusa nel JDK che mette a disposizione diverse funzionalità per la crittografia.

5. Analisi prestazionale del plugin Torrent

Una parte di questa tesi prevedeva l'analisi prestazionale del plugin Torrent mettendolo a confronto in termini di velocità di download, connessioni aperte, carico di CPU, occupazione di memoria e tempo di download rispetto ad alcuni client presenti sulla rete, tra i quali ad esempio Vuze, μTorrent, Trasmission, Tixati. Purtroppo non è stato possibile effettuare questa comparazione perché il plugin non porta a termine un download o meglio effettua un download incompleto rimanendo fermo al 99% e quindi lasciando il o i file incompleti, continuando a ricevere messaggi di 'keep alive'. Si è cercato di risolvere tale problema ma a tutt'ora non si è capito quale sia la causa dato che qualche volta riesce a terminare con esito positivo il download. Tuttavia non si potevano usare queste prove positive per gli scopi di questa tesi.
Si è comunque riusciti a fare una piccolo confronto con gli altri client relativamente al tempo di download e il plugin è risultato essere molto più lento in generale rispetto agli altri.

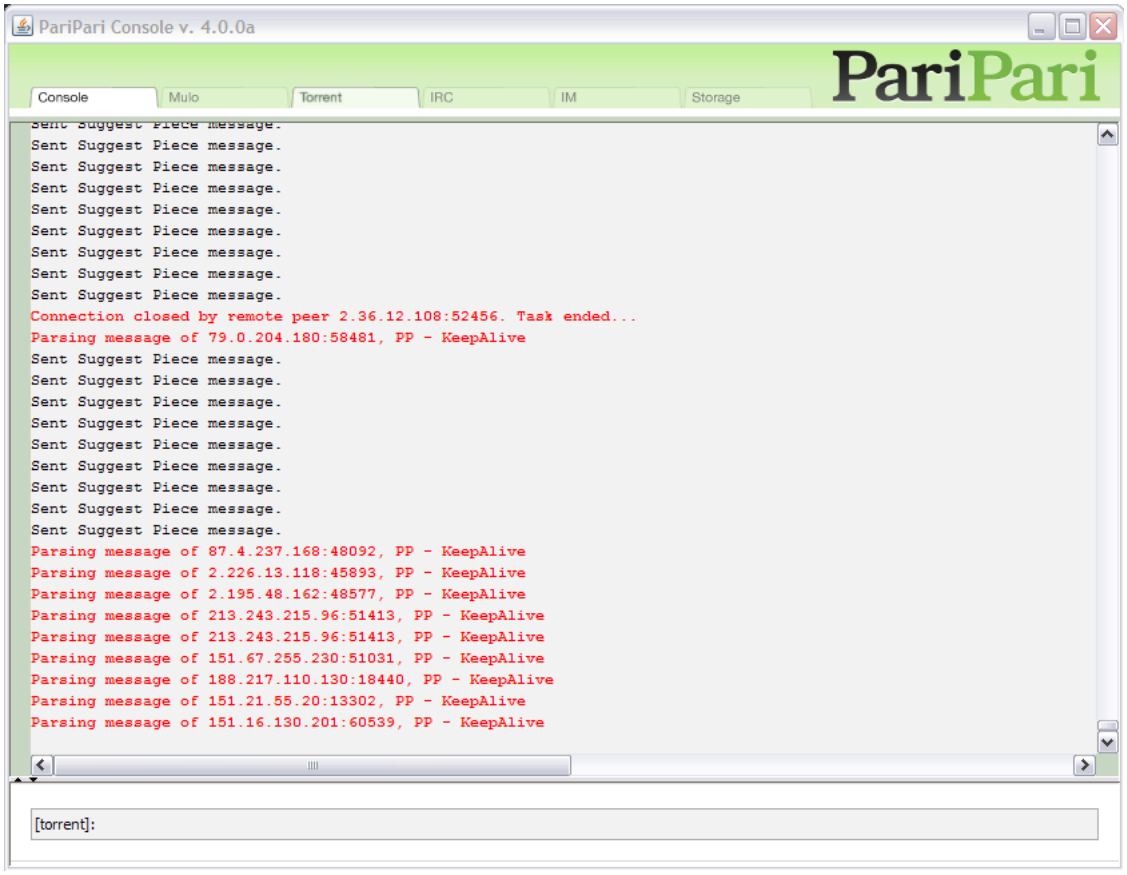

Figura 5.1.: Il download incompleto.

6. Conclusioni e stato dell'arte

Purtroppo non si è potuto paragonare Torrent con gli altri client a causa del problema rilevato ma si sono risolti altri problemi che c'èrano: non si poteva scaricare file di dimensioni al di sopra dei 2GB, cosa molto limitativa al giorno d'oggi che ormai si tende a scaricare file a partire dai 4GB in sù e si è reso accettabile il sovraccaricamento di CPU e RAM che rendeva inutilizzabile il plugin ma può essere ancora migliorato.
La spiegazione mediamente approfondita che si è data in questa tesi sia del protocollo BitTorrent che della sua implementazione nel plugin Torrent servirà di aiuto per i futuri torrentisti che vorranno affiancarsi a noi per rendere migliore il plugin e per avere uno spunto dal quale iniziare a creare un manuale completo del plugin.
Al termine di questa tesi sarà necessario dapprima risolvere il problema riscontrato e dopodichè sarà da fare una pulizia interna delle classi inutilizzate perchè creano solo confusione. Poi effettuare un merge con alcuni branch che sono rimasti e, cosa molto importante, fare molti molti test.
Quest'anno due ragazzi hanno aggiunto al plugin una nuova funzionalità: la possibilità di effettuare dall'interno del plugin la ricerca del file torrent.
Numerose sono ancora le features da aggiungere ma è necessario, a parer mio, rendere alquanto funzionante il plugin prima di implementare nuove funzioni.
C'è ancora molta strada da percorrere per poter avere un plugin che possa riuscire a competere, anche solamente in parte, con gli altri client presenti sulla rete.

A. BEncode

Il BEncode è una codifica usata da BitTorrent per memorizzare e trasmettere liberamente dati strutturati. Esso supporta quattro diversi tipi di valori :

- stringhe di byte
- interi
- liste
- dizionari

Anche se meno efficiente di una codifica binaria, la codifica BEncode è semplice e non è affetta dallo standard endian, che è importante per un programma multi-piattaforma come BitTorrent.

A.1. Interi

Un intero è codificato come i<numero in notazione base 10>e. Gli zeri davanti non sono permessi, anche se il numero zero e rappresentato come 0. I valori negativi sono codificati facendo precedere un segno meno al numero. Ad esempio il numero 42 sarà codificato come i42e, 0 come i0e, e -42 come i-42e. Lo zero negativo non è permesso.

A.2. Stringhe

Una stringha, che non necessariamente è composta da caratteri ma anche da byte, è codificata come <lunghezza>:<contenuto>. La lunghezza è codificata in notazione base 10, come gli interi, ma deve essere non negativa; il contenuto sono solo i byte che creano la stringa. La stringa "spam" sarà codificata come 4:spam. La specifica non tratta la codifica dei caratteri al di fuori dell codice ASCII; per mitigare a questo, alcune applicazioni BitTorrent comunicano esplicitamente la codifica (molto comune è la UTF-8) in vari modi non standard.

A.3. Liste

La lista di valori è codificata come i<contenuto>e. Il contenuto consiste nella codifica BEncode degli elementi della lista, in ordine, concatenati. Una lista consistente una stringa "spam" e il numero 42 sarà così codificata: i4:spami42ee. Da notare l'assenza di un sepratore tra gli elementi.

A.4. Dizionari

Un dizionario viene codificato come d<contenuto>e. Ciascuno degli elementi del dizionario sono codificati con una chiave seguita dal suo valore. Tutte le chiavi devono essere stringhe e devono apparire in ordine lessicografico. Un dizionario che associa i valori 42 e "spam" alle chiavi "foo" and "bar", rispettivamente, saranno codificati così: d3:bar4:spam3:fooi42ee (può risulatre più semplice la lettura se si aggiungono degli spazi d 3:bar 4:spam 3:foo i42e e). Non ci sono restrizioni su quali tipi di valori possono essere presenti nel dizionario e nelle liste; posso addirittura contenere altre liste e dizionari. Questo permette la codifica arbitraria di strutture dati complesse.

Bibliografia

[1] Bram Cohen, *The BitTorrent Protocol Specification*
http://www.bittorrent.org/beps/bep_0003.html, 2009.

[2] *Bittorrent Protocol Specification v1.0*
http://wiki.theory.org/BitTorrentSpecification.

[3] D. Turchetto, *PariPari Torrent: libTorrent and Fast Extension*, 2009.

[4] David Harrison, Bram Cohen, *Fast Extension*
http://www.bittorrent.org/beps/bep_0006.html, 2008.

[5] Arvid Norberg , Ludvig Strigeus, Greg Hazel, *Extension Protocol*
http://www.bittorrent.org/beps/bep_0010.html, 2008.

[6] *BitTorrentPeerExchangeConventions*
http://wiki.theory.org/BitTorrentPeerExchangeConventions.

[7] *azureus-protocol.txt*
https://trac.transmissionbt.com/browser/trunk/doc/azureus-protocol.txt?rev=2795, 2007.

[8] *Azureus messaging protocol*
http://wiki.vuze.com/w/Azureus_messaging_protocol, 2010.

[9] D. Turchetto, *Paritorrent: Seeding strategies*, 2011.

[10] S. Pozzobon, *PariPari: Modulo Torrent,* 2009.

[11] A. Dal Corso, *Paritorrent: Routing DHT,* 2011.

[12] *Extension negotiation protocol*
http://wiki.vuze.com/w/Extension_negotiation_protocol, 2010.

[13] A. Gallo, *Paripari: Crittografia Torrent* - 2009

[14] *PariPari Javadoc for Developers*
http://verona.dei.unipd.it/redmine/embedded/paripari/overview-summary.html

[15] IPv4, http://it.wikipedia.org/wiki/IPv4

Elenco delle figure

Elenco delle tabelle

Elenco dei codici